جامعة الانبار

كلية علوم الحاسوب وتكنولوجيا المعلومات

قسم أنظمة شبكات الحاسوب

برمجة كيانية OOP

المرحلة الثانية

الفصل الدراسي الأول والثاني

مدرس المادة

م.د. سميه عبدالله حمد

Dr. Sumaya Abdulla Hamad ٢٠٢٢ -٢٠٢٣

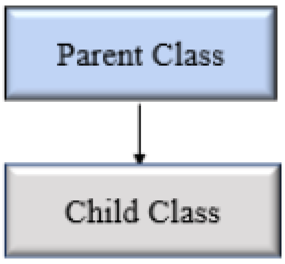
Types Of Inheritance

In Python, based upon the number of child and parent classes involved, there are five types of inheritance. The type of inheritance are listed below:

1. Single inheritance
2. Multiple Inheritance
3. Multilevel inheritance
4. Hierarchical Inheritance
5. Hybrid Inheritance

Single Inheritance

In single inheritance, a child class inherits from a single-parent class. Here is one child class and one parent class.



Python Single Inheritance

Example

Let's create one parent class called Vehicle and one child class called Car to implement single inheritance.

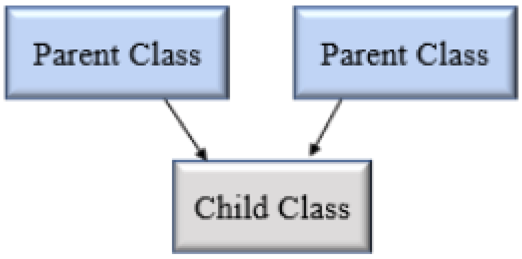
```
# Base class
class Vehicle:
    def Vehicle_info(self):
        print('Inside Vehicle class')
# Child class
class Car(Vehicle):
    def car_info(self):
        print('Inside Car class')
# Create object of Car
car = Car()
# access Vehicle's info using car object
car.Vehicle_info()
car.car_info()
```

Output

```
Inside Vehicle class
Inside Car class
```

Multiple Inheritance

In multiple inheritance, one child class can inherit from multiple parent classes. So here is one child class and multiple parent classes.



Python Multiple Inheritance

Example

```python
# Parent class 1
class Person:
    def person_info(self, name, age):
        print('Inside Person class')
        print('Name:', name, 'Age:', age)
# Parent class 2
class Company:
    def company_info(self, company_name, location):
        print('Inside Company class')
        print('Name:', company_name, 'location:', location)
# Child class
class Employee(Person, Company):
    def Employee_info(self, salary, skill):
        print('Inside Employee class')
        print('Salary:', salary, 'Skill:', skill)
# Create object of Employee
emp = Employee()
# access data
emp.person_info('Jessa', 28)
emp.company_info('Google', 'Atlanta')
emp.Employee_info(12000, 'Machine Learning')
```

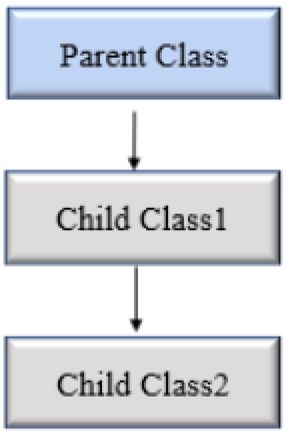
Output

```
Inside Person class
Name: Jessa Age: 28
Inside Company class
Name: Google location: Atlanta
Inside Employee class
Salary: 12000 Skill: Machine Learning
```

In the above example, we created two parent classes Person and Company respectively. Then we create one child called Employee which inherit from Person and Company classes.

Multilevel inheritance

In multilevel inheritance, a class inherits from a child class or derived class. Suppose three classes A, B, C. A is the superclass, B is the child class of A, C is the child class of B. In other words, we can say a **chain of classes** is **called multilevel inheritance.**



Python Multilevel Inheritance

Example

```
# Base class
class Vehicle:
    def Vehicle_info(self):
        print('Inside Vehicle class')
# Child class
class Car(Vehicle):
    def car_info(self):
        print('Inside Car class')
# Child class
class SportsCar(Car):
    def sports_car_info(self):
        print('Inside SportsCar class')
# Create object of SportsCar
s_car = SportsCar()
# access Vehicle's and Car info using SportsCar object
s_car.Vehicle_info()
s_car.car_info()
s_car.sports_car_info()
```

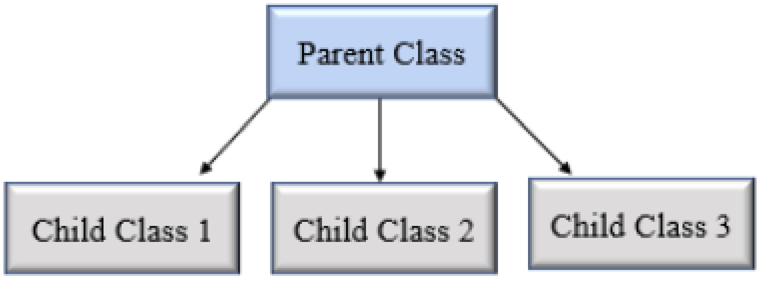
Output

```
Inside Vehicle class

Inside Car class

Inside SportsCar class
```

In the above example, we can see there are three classes named Vehicle, Car, SportsCar. Vehicle is the superclass, Car is a child of Vehicle, SportsCar is a child of Car. So we can see the **chaining of classes**.

Hierarchical Inheritance

In Hierarchical inheritance, more than one child class is derived from a single parent class. In other words, we can say one parent class and multiple child classes.



Python hierarchical inheritance

Example

Let’s create ‘Vehicle’ as a parent class and two child class ‘Car’ and ‘Truck’ as a child class.

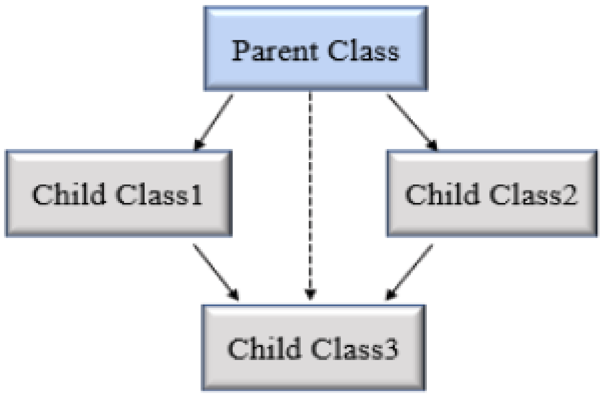
```
class Vehicle:
    def info(self):
        print("This is Vehicle")
class Car(Vehicle):
    def car_info(self, name):
        print("Car name is:", name)
class Truck(Vehicle):
    def truck_info(self, name):
        print("Truck name is:", name)
obj1 = Car()
obj1.info()
obj1.car_info('BMW')
obj2 = Truck()
obj2.info()
obj2.truck_info('Ford')
```

Output

```
This is Vehicle
Car name is: BMW
This is Vehicle
Truck name is: Ford
```

Hybrid Inheritance

When inheritance is consists of multiple types or a combination of different inheritance is called hybrid inheritance.



Python hybrid inheritance

Example

```
class Vehicle:
   def vehicle_info(self):
      print("Inside Vehicle class")
class Car(Vehicle):
   def car_info(self):
      print("Inside Car class")
class Truck(Vehicle):
   def truck_info(self):
      print("Inside Truck class")
# Sports Car can inherits properties of Vehicle and Car
class SportsCar(Car, Vehicle):
   def sports_car_info(self):
      print("Inside SportsCar class")
# create object
s_car = SportsCar()
s_car.vehicle_info()
s_car.car_info()
s_car.sports_car_info()
```

Output:

```
Inside Vehicle class
Inside Car class
Inside SportsCar class
```

Note: In the above example, **hierarchical** and **multiple** inheritance exists. Here we created, parent class Vehicle and two child classes named Car and Truck this is hierarchical inheritance.

Another is SportsCar inherit from two parent classes named Car and Vehicle. This is multiple inheritance.

Python super() function

When a class inherits all properties and behavior from the parent class is called inheritance. In such a case, the inherited class is a subclass and the latter class is the parent class.

In child class, we can refer to parent class by using the super() function. The super function returns a temporary object of the parent class that allows us to call a parent class method inside a child class method.

Benefits of using the super() function.

1. We are not required to remember or specify the parent class name to access its methods.
2. We can use the super() function in both **single** and **multiple inheritances**.
3. The super() function support code **reusability** as there is no need to write the entire function

Example

```
class Company:
    def company_name(self):
        return 'Google'
class Employee(Company):
    def info(self):
        # Calling the superclass method using super()function
        c_name = super().company_name()
        print("Jessa works at", c_name)
# Creating object of child class
emp = Employee()
emp.info()
```

Output:

```
Jessa works at Google
```

In the above example, we create a parent class Company and child class Employee. In Employee class, we call the parent class method by using a super() function.

issubclass()

In Python, we can verify whether a particular class is a subclass of another class. For this purpose, we can use Python built-in function issubclass(). This function returns True if the given class is the subclass of the specified class. Otherwise, it returns False.

Syntax

```
issubclass(class, classinfo)
```

Where,

- class: class to be checked.
- classinfo: a class, type, or a tuple of classes or data types.

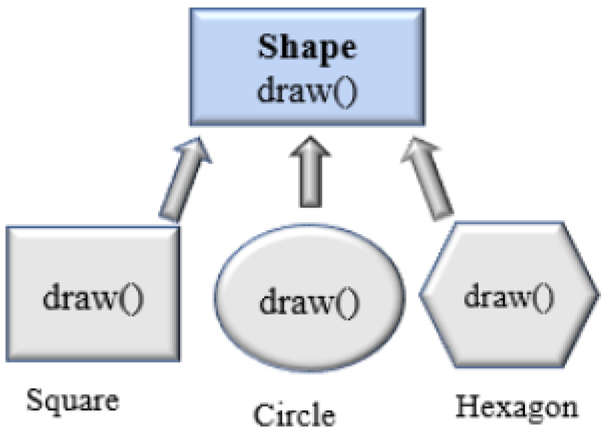
Example

```
class Company:
    def fun1(self):
        print("Inside parent class")
class Employee(Company):
    def fun2(self):
        print("Inside child class.")
class Player:
    def fun3(self):
        print("Inside Player class.")
# Result True
print(issubclass(Employee, Company))
# Result False
print(issubclass(Employee, list))
# Result False
print(issubclass(Player, Company))
# Result True
print(issubclass(Employee, (list, Company)))
# Result True
print(issubclass(Company, (list, Company)))
```

Method Overriding

In inheritance, all members available in the parent class are by default available in the child class. If the child class does not satisfy with parent class implementation, then the child class is allowed to redefine that method by extending additional functions in the child class. This concept is called **method overriding**.

When a child class method has the same name, same parameters, and same return type as a method in its superclass, then the method in the child is said to **override** the method in the parent class.



Python method overriding

Example

```
class Vehicle:
    def max_speed(self):
        print("max speed is 100 Km/Hour")
class Car(Vehicle):
    # overridden the implementation of Vehicle class
    def max_speed(self):
        print("max speed is 200 Km/Hour")
# Creating object of Car class
car = Car()
car.max_speed()
```

Output:

```
max speed is 200 Km/Hour
```

In the above example, we create two classes named Vehicle (Parent class) and Car (Child class). The class Car extends from the class Vehicle so, all properties of the parent class are available in the child class. In addition to that, the child class redefined the method max_speed().

Method Resolution Order in Python

In Python, Method Resolution Order(MRO) is the order by which **Python looks for a method or attribute**. First, the method or attribute is searched within a class, and then it follows the order we specified while inheriting.

This order is also called the Linearization of a class, and a set of rules is called MRO (Method Resolution Order). The **MRO plays an essential role in multiple inheritances as a single method may found in multiple parent classes**.

In multiple inheritance, the following search order is followed.

1. First, it searches in the current parent class if not available, then searches in the parents class specified while inheriting (that is left to right.)
2. We can get the MRO of a class. For this purpose, we can use either the mro attribute or the mro() method.

Example

```
class A:
    def process(self):
        print(" In class A")
class B(A):
    def process(self):
        print(" In class B")
class C(B, A):
    def process(self):
        print(" In class C")
# Creating object of C class
C1 = C()
C1.process()
print(C.mro())
# In class C
# [<class '__main__.C'>, <class '__main__.B'>, <class '__main__.A'>, <class 'object'>]
```

In the above example, we create three classes named A, B and C. Class B is inherited from A, class C inherits from B and A. When we create an object of the C class and calling the process() method, Python looks for the process() method in the current class in the C class itself.

Then search for parent classes, namely B and A, because C class inherit from B and A. that is, C(B, A) and always search in **left to right manner.**